GOTHIC memory management : a multiprocessor shared single level store

Béatrice Michel

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.(1)39 63 55 11

Rapports de Recherche

N° 1202

Programme 3
Réseaux et Systèmes Répartis

GOTHIC MEMORY MANAGEMENT : A MULTIPROCESSOR SHARED SINGLE LEVEL STORE

Béatrice MICHEL

Avril 1990

RR-1202

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

GOTHIC Memory Management: a Multiprocessor Shared Single Level Store

La Gestion Mémoire de GOTHIC : un Stockage Uniforme des Données avec Partage sur Multiprocesseur[†]

Béatrice MICHEL[‡]

Publication Interne n° 523 - Mars 1990 - 20 Pages

[†]Ce travail a été accompli à l'IRISA, dans l'équipe *Langages et Systèmes Parallèles* dans le cadre du projet INRIA/BULL Gothic

[‡]Adresse actuelle de l'auteur : Princeton University - Department of Computer Science, 35 Olden Street, Princeton, New-Jersey 08544-2087

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Abstract: Gothic purpose is to build an object-oriented fault-tolerant distributed operating system for a local area network of multiprocessor workstations. This paper describes Gothic memory manager. It realizes the sharing of the secondary memory space between any processes running on the Gothic system. Processes on different processors can communicate by sharing permanent information. The manager implements a shared single level storage with an invalidation protocol working on disk-pages to maintain shared data consistency. The result is a secondary memory shared through a local area network of loosely coupled multiprocessor workstations. This memory manager has been implemented in the current version of the Gothic operating system prototype. It is particularly suitable for distributed applications with relatively local calculations and references, such as distributed data base systems.

Key words: Gothic, multiprocessor, distributed system, memory management, shared data consistency, single level store.

Résumé: L'objectif du projet Gothic est de réaliser un système distribué, orienté objet, tolérant les fautes, s'exécutant sur un réseau de machines Bull SPS7 multiprocesseurs. Ce rapport décrit la conception et la réalisation du gestionnaire de la mémoire de Gothic, qui réalise la mise en commun de la mémoire secondaire entre tous les processus Gothic. Ceci permet aux processus de communiquer par partage d'information permanente. Le gestionnaire de la mémoire réalise un stockage uniforme des données, et un protocole à invalidation est implémenté pour maintenir cohérente toute page d'information partagée. C'est avec une telle gestion de la mémoire que fonctionne le prototype actuel du système Gothic. Cette gestion est appropriée pour toute application présentant une localité des calculs et des références, comme en particulier les bases de données réparties.

Mots clés: Gothic, multiprocesseurs, systèmes distribués, gestion mémoire, cohérence des données partagées, stockage uniforme des données.

1 Introduction

Gothic is a joint INRIA/BULL S.A. research project which aims at building an object-oriented fault-tolerant distributed operating system for a local area network (a LAN) of multiprocessor workstations (MW) [BANA 88a].

Distributed computer systems seem to be able to meet many users' requirements, such as those about distributed computation, large storage capacity, and reliability. Those systems, built from many small or medium computer workstations connected by a network, are relatively cheaper than tightly coupled multiprocessors or mainframes. They can be geographically distributed. Distributed systems are really interesting by the sharing of resources they can support. They can be managed so that some computer's devices are made available to anyone in the network. A resource such as a laser printer can be managed by the computer which owns it so that any user in the network can reach and use this device, the cost of the device being divided between all the users. Sharing of more basic devices, such as computing power and/or memory, can also be implemented in those systems. The idea is to build a *common pool* of processors, of memories, and to allocate them to any user according to his/her needs. Sharing devices and resources has many advantages but brings also some problems: contention, fair allocation between users, shared data consistency, ... But this sharing of memory and/or of computing power realizes data and/or calculus distribution between computers, which really answers users' requirements. The Gothic project is investigating how to realize these distributions, and already provides the sharing of the secondary memory storage on a network of multiprocessor workstations.
Distributed systems can also be managed to present good features towards the fault-tolerance issue. First, provided it is "well-built", the operating system easily adapts to another computer/workstation plugged in/out the network. Second, each computer can be quite autonomous. Fault-tolerance at the hardware level is closely related to redundancy. Some systems, such as Tandem[BART 81], achieve their fault-tolerance purpose by doubling any component in the system, bus, memory, cpu, etc. Gothic's concern about fault-tolerance is not to build a whole fault-tolerant machine, but to provide fault-tolerant process executions. A special stable storage memory board has been designed and built for this purpose and is added to any Gothic processor. Actually, Gothic's researchs about fault-tolerance are joining transactionnal systems. Reliable communication primitives are also implemented using these special memory boards.
Gothic includes both operating system and language researchs. Gothic's language is an object-oriented one. This feature will be integrated down into the system. Some kinds of capabilities will be implemented, ensuring high protection on data.

Gothic's memory management is discussed in this paper. It is explained in detail in [MICH 89]. Beyond the scope of this paper are Gothic's results about fault-tolerance, reliable communications, and research to express data and calculation distribution with new language concepts. Interested readers should consult [BANA 88b], [MULL 88] for fault-tolerance issues, [BANA 89] for reliable communications concerns, and [BANA 86] about language and

expression of distribution.

Gothic's memory manager realizes the sharing of the secondary memory space between any processor in the system. Information storage is done by a single level storage (a SLS). Any Gothic permanent information is managed and kept consistent as a common memory is in a tightly coupled multiprocessor. This management of memory in a loosely coupled multiprocessor is quite new, is very convenient for process communication, and is very useful to achieve calculus distribution or parallel clause execution. When designing Gothic memory management we were deeply interested in Apollo Domain [LEVI 86] for the shared SLS implementation. Many things done were suggested from the study of this system. However Apollo files sharing policy is different from Gothic. Apollo's consistency protocol works with a file granularity and is a lock/unlock protocol. Moreover Apollo's protocol keeps files consistent only within a node and not through the whole network. Gothic keeps information pages consistent through the whole network. And if Gothic's granularity is too fine for an application, some tools (semaphores) are available to implement the needed coarser granularity. Ivy [LI 86] is a system implementing a shared virtual memory on a loosely coupled multiprocessor. Although Gothic's protocol works on disk-pages whether Ivy's works on virtual pages, Gothic consistency management protocol principles are quite similar to Ivy's. But it is important to note that it is the virtual addressing space which is common and shared in Ivy, whether it is each permanent disk-information which can be shared in Gothic. Sharing does not occur at the same level in the memory hierarchy. Gothic does not restrict the sharing and the consistency management to processes using the same addressing space. Clouds [RAMA 88] is a research project which presents common points with Gothic. Gothic consistency requirement and protocol are different however. Cloud's protocol works with an object granularity.

The following of this paper is organized in five sections. The second section gives an overview of Gothic since this project is the context in which this work is done. The third section presents general issues about permanent information management such as access and storage issues. The fourth section deals with information sharing in Gothic; it explains and discusses the protocol maintaining shared data consistency. Section five briefly describes the implementation steps followed to obtain the Gothic system prototype. The conclusion presents some measurement, and Gothic future.

2 Gothic Architecture

2.1 Machine Architecture

Gothic target architecture is a local area network (LAN) of Bull SPS7 multiprocessor workstations (MW) [BULL 85a]. Figure 1 shows one of the simplest configurations of a Gothic system, in which two SPS7 workstations are connected by an Ethernet nerwork.

Each SPS7 workstation has two 68020 processor, each with a local memory of 4 up to 16 M-

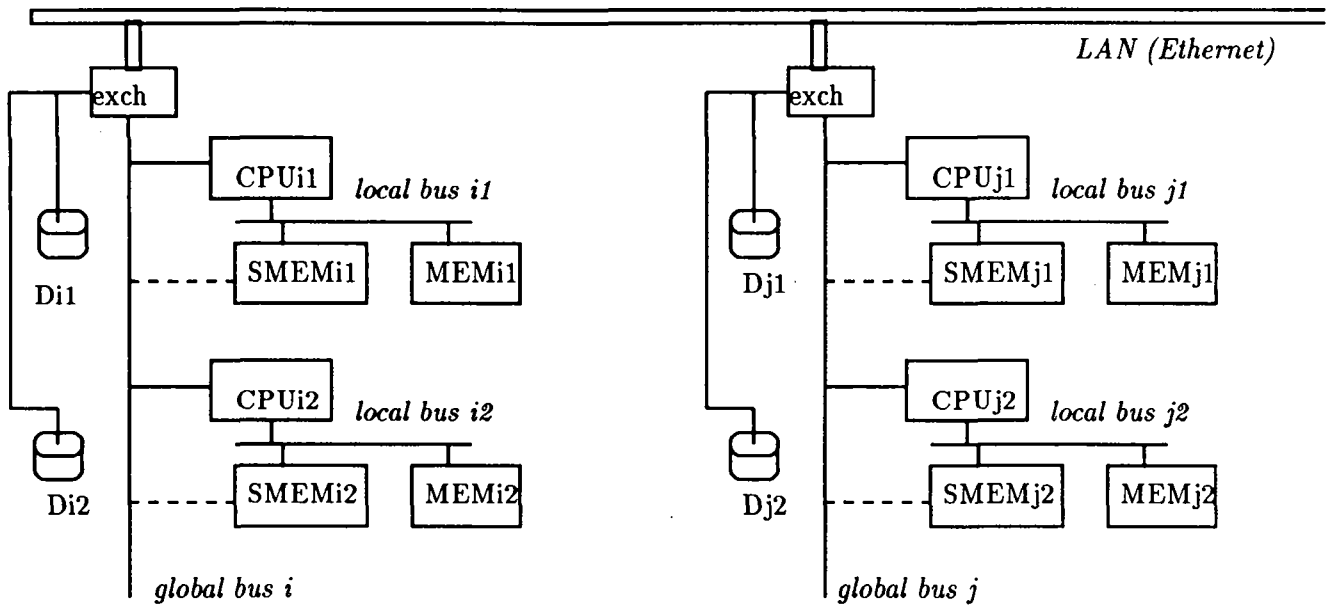

Figure 1: Example of an Architecture for a Gothic System

bytes, and a stable storage memory board for fault-tolerance purpose. Processors communicate which each other through message passing.

2.2 System Architecture

The Gothic system architecture consists of five levels as depicted in Figure 2.

5	Applications	
4	Polygoth	
3	Reliable Communication Primitives	
2	Gothic Kernel	
1	Bull SPS7 multiprocessor workstations	Stable Storage Memory Boards

Figure 2: Gothic Organization

The hardware level at the bottom is the Bull SPS7 [BULL 86] multiprocessor workstations plus the stable storage memory boards developed at IRISA. Next level is the Gothic kernel. Basic entities are segment and process. The whole memory management uses the segment as a basic entity. Process is used to express and manage a program execution on the Gothic system.

The kernel includes management of peripheric devices, communication, process, memory, and so on. The work described in this paper, i.e.: Gothic's memory manager, belongs to this level. The third level consists of reliable communication primitives for rendezvous and parallel clause atomic communications. The fourth level is the *language level* which includes studies to express parallelism, parallel clause nestings, and data and calculation distribution [LECL 89], [LECE 88]. Applications are at the fifth level.

This paper discusses in detail the design and implementation of Gothic memory management. Gothic aims at being an object-oriented system running on a LAN of MW where information can be accessed and shared by any process no matter the processor on which the process is currently running. Gothic memory management has to be convenient for parallel clause executions. Any information used on the Gothic system is a permanent one. Gothic memory management, therefore, is very much concerned with permanent information management issues.

3 Permanent Information Management Issues

In Gothic any piece of information belongs to a permanent *segment.* A segment is the set of memory pages storing a whole logical piece of information. Managing these segments implies to consider naming, localization, access and storage issues (see Figure 3). For Gothic special purposes, segment sharing is also to be studied; this will be presented in the next section.

External naming in Gothic is a Unix-like one, where names are built according to a hierarchical tree-organization. Gothic external names look like: "/dir1/dir2/.../dirn/segment-name". This hierarchical tree-organization is only a logical one. It is possible for two segments to belong to the same directory and to be stored on different (physical) disks.

Universal Unique IDentifiers (UUID) are used to give any Gothic segment an internal name. A segment's UUID is built from two integers: the date this segment was created and the number of the processor where this segment was created. Such UUID are location independent; and as long as the migration mechanism itself is not considered but just migrated segments, segment migration does not ask for further requirement than a localization protocol. When a processor P needs to locate a segment S, two situations can arise:

- 1st case: processor P never accessed S before. A message is sent to the processor where S was created and whose name is stored in S's UUID. If this creator processor does not own S anymore then P broadcasts a *localization message* to any other processor in the system. The processor which owns S then responds so that P knows S's localization. If nobody responds, either S has been destroyed, or S never existed; anyway, it is an error case.
- 2nd case: processor P has accessed S previously. P has memorized the identity of the processor P' which owned S at this time. P asks P' for S; if S has migrated since, P' is

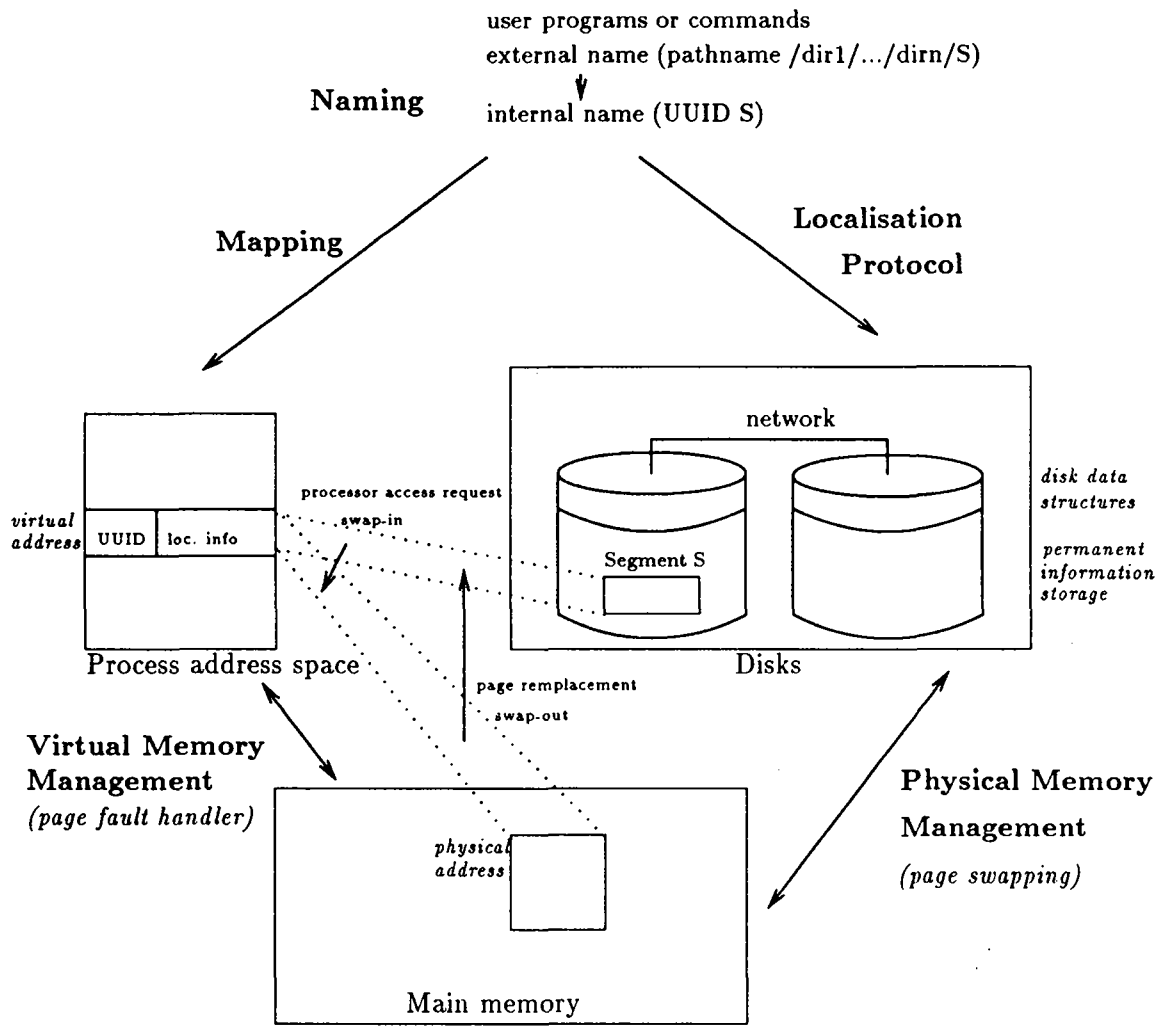

Figure 3: Permanent information management issues

no more S's owner. Then P broadcasts a localization message which is handled as in the first case.

This broadcast localization message is somewhat "heavy" for the system because every processor has to respond. However, we choose this approach because it is simple, and segment migrations are rare. This protocol is improved by using an *hint file* which keeps information about migrated segments.

3.1 Access to Gothic Segments

For the processor to work on a permanent information, this information has to be brought from the disk to the processor main memory. This swapping can be implemented by a *virtual memory mechanism* (a VMM). There are many different VMM. A brief description of a *paged virtual memory mechanism* is given now since Gothic implements this mechanism. A *page* is a fixed size piece of memory; a page size in Gothic is 1 K-byte. A paged virtual memory uses the page as its basic unit for the whole memory management. In Gothic, it is the page unit and not the segment unit which is used in the lowest levels of the memory management. Main memory is viewed and managed as a set of pages by the operating system. The VMM brings information from disk to main memory pages after pages when the processor asks to access them. The system uses an array data structure to manage main memory pages, to know which pages are having a copy in main memory and which main memory pages are free. A processor wanting to access a data stored in a "Py" page of a "UUIDx" segment is in one of these two situations:

- This page has been copied in one of the processor's main memory page. The processor can access it immediately.
- This page is not in the main memory, it is a page fault case. The operating system tries to solve it and calls the page fault handler. It first has to localize the faulting page (localization of the segment this page belongs to; this localization protocol has been described previously). Then this page must be copied into a main memory page. This is easy as long as there is at least one free page in main memory. However, when there is no room left, a page remplacement algorithm is called for deciding upon which page can be freed and used to copy a new page in. Many page remplacement algorithms exist, following different strategies (lru, lfu, fifo, at random, ...). They all try to determine with the highest probability pages that the processor is not going to ask to access in the near future. But the purpose of this paper is not to discuss these strategies.

To summarize, a VMM controls and does the swap-in/out of information between main memory and secondary storage devices. It makes permanent informations accessible to the processor.

How is translated the virtual address a process gives to the processor into the physical address of the information? Each process is given by the system an array when created called this process's addressing space. The system uses this array to memorize which information this process has asked for and is allowed to access: before accessing an information a process has to ask the system for mapping this information into its addressing space; if this access is allowed an information descriptor is put into the process's addressing space and the process can then access the information through virtual addresses. The system translates virtual addresses according to the process's addressing space and is therefore able to compute corresponding physical addresses of the information. (The information can then be found in main memory, if the VMM has allready brought it here, or on disk). A process can access an information only after the system has done the mapping of this information segment into its addressing space.

3.2 Information Storage

There are two ways to store and access permanent informations [HOUD 81].

- The first one is the "classical" one, with two addressing modes. To access permanent files users call explicit input/output primitives: read-file, write-file, for example. Buffers are used to transfer the information from/to disk. To access informations in main memory, they use virtual addressing. There are two representations of information, depending on whether the information is copied into main memory (by the VMM), where it is represented as segments, or if it is stored on disks, and represented then as a permanent information, as a file. Main features of this approach are the two addressing and accessing modes, and the different information representations in main memory and on disk. So information has to be copied and "transformed" when brought from disk to main memory. There are two access levels to information.
- The second one is called the *Single Level Storage* approach. The virtual addressing mode is used to access information both in main memory and on disk. Users access any information with this "transparent" virtual addressing. There is also a unique way to represent information. Information does not have to be transformed when copied into main memory, and the virtual addressing mode is the only one. The swapping is performed directly between the copy of the information in the processor main memory and the information stored on disk.

Because of these features and simplicity it is this second solution which is implemented for Gothic information storage. Information in Gothic is represented and managed as sets of pages as well in main memory than on disk.

3.3 Summary about Gothic Permanent Information Management

Gothic memory management is a paged VMM working with a Single Level Storage (SLS). For Gothic's purposes, it realizes the sharing of disks: it is a generalized SLS. Swap in and out do not work only between a processor's main memory and the disk this processor is managing, but between its main memory and the whole set of disks managed in the Gothic system (network wide swapping). Any process no matter the processor on which it is running can access any information no matter the disk on which this information is stored, provided this process is having sufficient access rights to this information. To processes disks are a *virtual disk*, a shared resource for everyone in the system. Processes can communicate by sharing permanent information. Before the next section about information sharing, the four interface primitives to Gothic's memory management are given:

- create-segment(out:UUID): Called by a process to create a segment on disk. This new segment's uuid is the output parameter.
- delete-segment(in:UUID): Called by a process to delete on disk the segment which uuid is the input parameter.
- access-segment(in:UUID, in:access-mode, out:virtual-address): Called by a process when it wants the 'UUID' segment to be mapped into its addressing space. The output parameter is the virtual address of this segment in the process's addressing space.
- deaccess-segment(in:UUID): Called by a process when it wants to unmap the 'UUID' segment out of its addressing space.

4 Information Sharing in Gothic

Disks in a Gothic system are managed as a *virtual disk*, a resource shared between all processors in the system. Each processor uses its local memory as a cache of the virtual disk. However, sharing disks and allowing multiple copies of information bring in consistency problems. A consistency maintenance protocol is required to manage sharing according to the chosen sharing policy. Gothic information consistency definition is the following:

> *An information is consistent if the value returned by a read operation is the same as the value written by the most recent write operation to the same address [CENS 78].*

Actually, Gothic manages each permanent segment as a common main memory is managed in a tightly coupled multiprocessor.

Gothic memory manager does not consider the architecture as made of workstations but rather as made of sites. A site is the set composed by a processor, its local memory, its stable storage, and disk partition(s) managed by this processor. There are four sites on Figure 1. Gothic operating system manages the architecture on which it is running as many sites connected by a communication link (bus or network).

Figure 4 shows a consistency example. Two processes, p2 and p3, running respectively on processors CPU2 and CPU3, are accessing a same page of information. The VMM brings a

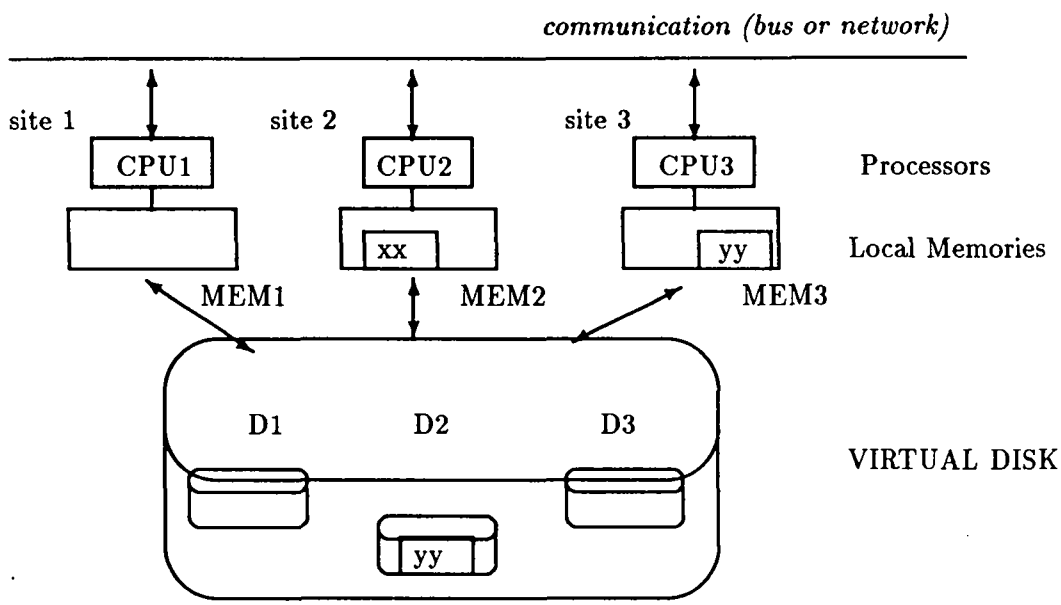

Figure 4: Example of a Consistency Problem

copy of this page in both CPU2's and CPU3's local memory. Everything goes smoothly as long as p2 and p3 just read the page. If p2 for example, modifies this page (it writes xx), then there are different versions of the same page of information in the system. More precisely, in the case drawn on Figure 4, CPU2's local memory copy(xx) is different from CPU3's local memory copy(yy) and from the page stored on disk(yy). It belongs to the consistency protocol to make everybody knows that the accurate version of this page, according to the chosen consistency definition, is the one in CPU2's local memory.

How does Gothic consistency protocol work? The consistency management protocol implemented ensures consistency of any shared word of information. However, for performance reasons and because of the target architecture's hardware, the word consistency wanted is implemented by a page consistency, a page size being 1K-byte. It is an invalidation protocol which allows many readers or only a writer on the same page at the same time. Any access to any piece of information is checked for consistency. Each segment has a home associated with a disk. The site that manages the disk maintains the consistency of the segment pages. This site is the only one that knows the exact mode of the pages and which sites have copies. Other sites know their local access right on their copy of the pages. They cooperate with the manager site to maintain the consistency. Any page can be in one of the following modes:

- invalid: There is no copy of this page in any local memory of any site (the page is just stored on disk).
- write: Only one site has a copy of the page with an exclusive access right.

- read: There can be multiple copies of the page in the system. No processor can modify it without causing a write page-fault.

Any access of a processor to a page leading to a page-fault has to be considered by this page's manager site and is checked for consistency there. If there are several page faults occurring on a same page at the same time, their handlers are serialized on the site managing this page.

The following scenario sets out how consistency is kept while page faults are solved. This scenario takes place on the system drawn on Figure 5. (About data structures used: the lock field is used to sequentialize different page-fault handlers asking for the same page at the same time. A lock is required on any site to serialize page faults on the same page within a site: Local-Table(UUID,p).lock, and on the manager site to serialize all the page faults on the same page in the whole system: Manager-Table(UUID,p).lock).

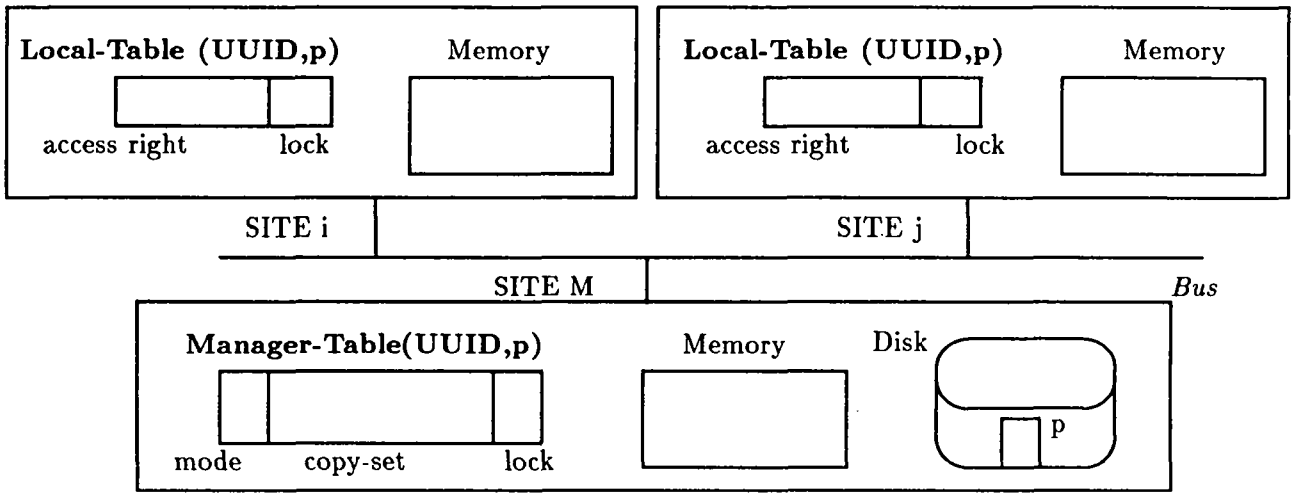

Figure 5: Gothic Consistency Management Protocol (data structures used)

- 1st step: There are three sites in this Gothic system. The sharing of a page of information called 'p' is considered. Site M is the manager of p while sites i and j are just "ordinary" sites to p. First p is *invalid* as the only instance of p is the one stored on M's disk.

- 2nd step: Now, a process running on site j needs to read p. A read page fault then occurs on j. Site M (p's manager) is asked for solving it. M checks in its table and sees that as nobody is owning a copy of p there is no problem for sending one to j. M memorizes j is now having a copy of p in read mode: p.copyset = j, p.mode = read. j keeps that it is owning a copy of p in its local memory with a read access right, too.

- 3rd step: Now another process running on site i also wants to read p. Another read page fault occurs, but on site i. M has to solved it. M sees in its table that j is owning a copy

of p in read mode. As many readers are allowed on the same page at the same time (it does no present any danger for this page consistency) M asks j to send i a copy of p; then M adds i to p.copyset; p.mode remains the same (read).

- 4th step: The process running on j now wants to write p. A write page fault occurs. M looks in its table to see which other sites are having a copy of p, and sees that just i has one. M sends then an invalidation message to i for i to invalidate its copy of p. Afterwards, j becomes the only site having a copy of p and is given an exclusive access to p: a write access. M updates its record about p, so p.copyset = j, and p.mode = write. p consistency is maintained.

Gothic memory management provides consistent information sharing between any Gothic processes no matter the processor on which each one is running, and thus is suitable for parallel clause execution. Gothic protocol checks any word access for consistency. It is not a lock/unlock protocol, but an invalidation protocol allowing several readers or one writer on the same page at the same time. Therefore, if an information is stored in more than a page, there can be many writers on the same information at the same time, each one on a page. This protocol could seem a very hardworker and heavy one. Actually it gives lots of work only when really needed: checking any access for consistency will cost only in case some invalidations are required. It is the case only if there are several processes sharing a page in write mode, and there are cases where this fine sharing is really required. In other cases this protocol does not cost more than an Apollo-like one or any other consistency protocol for loosely coupled multiprocessors. This checking of any word access for consistency is really needed and convenient for parallel applications, for data sharing, synchronization, communication, parameters passing. With such a system programming synchronization in a parallel clause can simply consist in checking the value of a shared variable.

Gothic consistency protocol has the finest possible granularity according to the target machine. But bigger ones can also be handled. The word granularity implemented through a page granularity is the consistency granted by the operating system. However, Gothic offers some facilities for implementing coarser granularities if required by an application. Other sharing policies can be implemented by using *Gothic semaphores*. These semaphores are special tools managed by Gothic operating system. An Apollo-like sharing policy with a segment granularity can be implemented by using these semaphores to lock and unlock process accesses to this segment. The user can also creates his/her own synchronization objects using Gothic sharable segments to implement his/her own sharing policy. However he/she must keep in mind that a ping-pong effect [DUBO 88] can then arise on these objects which cannot arise using Gothic semaphores.

Gothic protocol for maintaining a segment consistency is centralized on this segment's manager site. As any input and output to/from the disk segment has to be performed by the processor managing this disk, and are so centralized, it seems to be the best solution. Moreover, it makes the protocol rather easy to design and implement.

Another point that has to be precised is that each Gothic process is given its own addressing space. Pointers would have to be translated into (segment UUID, pointer inside this segment) when passing to another process. There is no lightweight process in Gothic today. In the actual prototype, process creation is costly, and so is process migration. A process's context is rather big, quite similar to a Unix process's and makes context switching not fast. Issues dealing with process definition, lightweight and heavyweight processes, process migration, are under study for Gothic's next version.

5 Implementation

This section briefly reviews implementation work carried out to obtain a Gothic system first prototype running on a MW (a Bull SPS7 MW). Three main steps compose this work. Two of them are completed today, and the third one is under achievement. The first step consists in the building of a Gothic system for a single processor workstation. The next step extends this system so that it runs on a MW. A final third step is adapting the system to a LAN of MW.

5.1 First Step

We start from a Bull S.A. operating system called Spart [BULL 87] which runs on the target Bull SPS7 workstation [BULL 85b]. This workstation has just one processor for this first step. Spart operating system serves as a basis for building Gothic, instead of starting from "nothing". It is adapted in order to become a Gothic system; actually, this really means to re-write it nearly completly, but progressively. This is the advantage of the method: each time something is modified in the system it can immediately be tested since any other system parts remain unmodified. Spart adaptation to a paged VMM and a SLS are the main problems in the achievement of this step. Spart indeed did not work with a real VMM. Spart was originally built to run on a 68010 processor with a segmented Memory Management Unit (MMU). There were no page faults. Spart was then adapted in a straightforward way to run on a 68020 processor but did not use all of this processor functions. It still worked with the same segmented MMU. The processor board is changed to a board with a 68020 processor and a MT68851 Paged MMU [MOTO 86]. To adapt Spart, page fault handlers are written, the whole main memory management is changed: main memory is now managed as a set of pages, and process addressing space management is rewritten. To implement a SLS, a permanent information management system is built to manage Gothic segments on disk. And a minimal external naming system is made for Gothic segments. This first step results in having an operating system working with a paged VMM and a SLS running on a processor. This step is the longest to achieve and asks for a lot of implementation, coding, and debugging works.

5.2 Second Step

The *virtual disk* implementation is this step main issue. It implies the implementation of the consistency management protocol. The aim is to have a system running on a MW where different processes can share a permanent segment, this permanent segment being kept consistent according to Gothic consistency definition. The architecture is divided into sites. Each site is given the management of the disk (or disk partition) which is associated to it. When a process running on a site wants to access a segment, the local disk manager first checks if this segment is stored on the local disk. If so, the whole access request is handled locally. Otherwise the access request is sent to the site managing the disk where this segment is stored; it is a Remote Procedure Call. The request is handled on this distant site and resulting parameters if any are sent back to the calling site (see Figure 6). The consistency management protocol described in the previous section is implemented as a part of this virtual disk implementation.

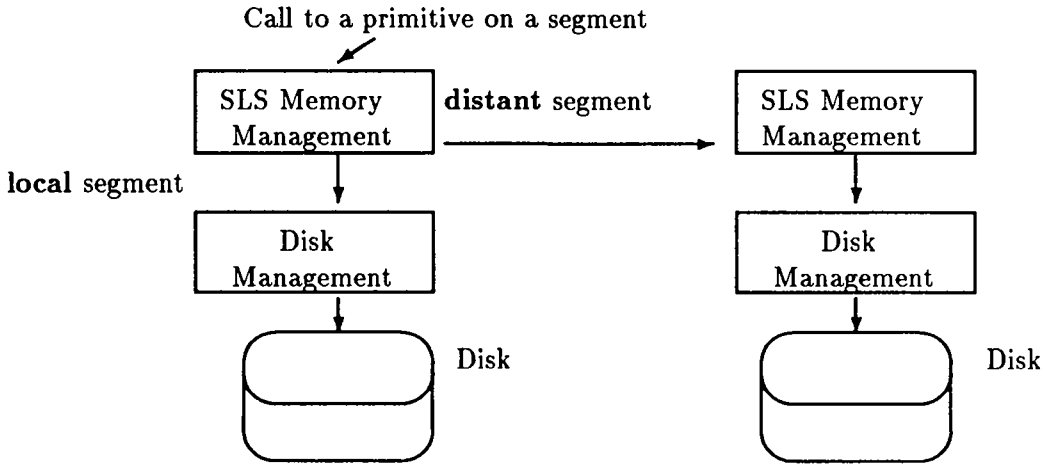

Figure 6: Virtual Disk Implementation

Gothic manages exceptions as the initial Spart system did. Any faulting process is aborted while an explicit message is displayed on the system console screen to inform the user which fault occurred. The code returned is composed of the number of the kernel primitive where the fault occurred and the error specific code. This exception management is also implemented in page fault handlers and any other primitives added to the initial Spart kernel.

Today a Gothic system is running on a Bull SPS7 MW. A paged MMU is implemented, segments are stored with a SLS, and shared segments are kept consistent.

6 Conclusion

6.1 Measurement

Some tests were done on this initial version of the Gothic system mainly to check the behaviour of the consistency management protocol (the burden it puts on program execution). The following Figure 7 shows results gained from a small test done for this purpose.

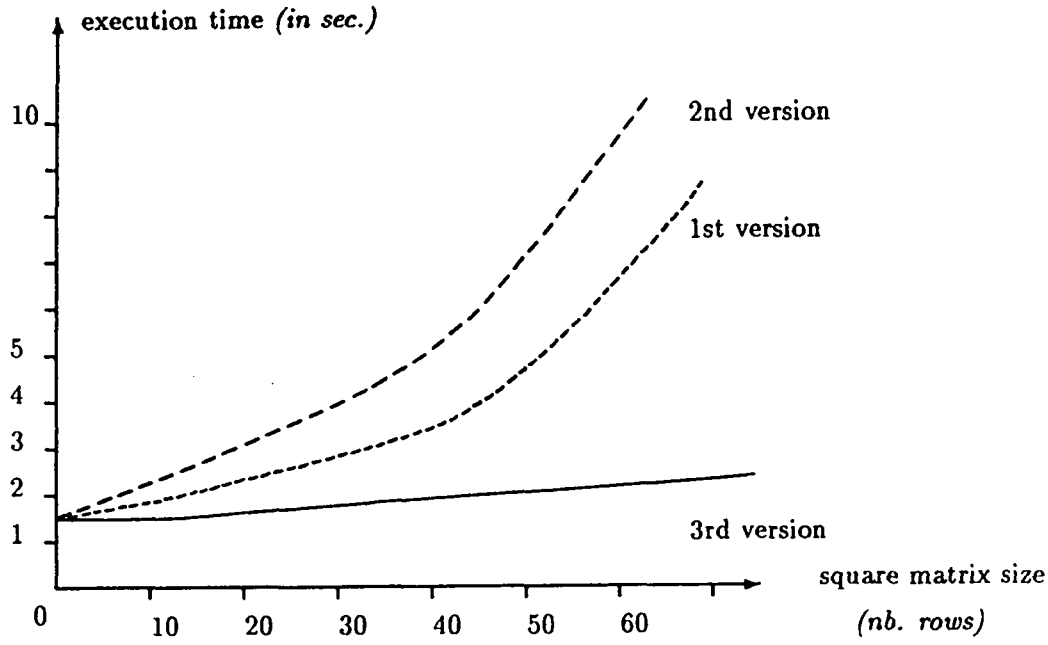

Figure 7: Measure on Square Matrix Product

The test program computes the product of two square matrix. This is performed in parallel by two processes running on two processors. A process calculates the resulting matrix odd row elements while the other calculates the even row elements. The two initial matrix are shared in read mode while the resulting matrix is shared in write mode. Gothic consistency protocol has to perform invalidations for keeping the latter consistent.

There are three versions of this program. In the first version, each time an element is totally computed it is written into the resulting matrix. The second version is the worst one because it leads to many invalidations: each time any operation is performed (any addition or product), the result is reported into the resulting matrix, even if it is only a step in a calculus and not the definitive value of an element. Of course, this leads to many write operations into the matrix, and therefore many invalidations. The third version is the best as it tries to reduce the number of writes into the resulting matrix and so reduces the number of invalidations: it waits to have computed all the elements of a row to write then the whole row into the resulting matrix.

This small test points out that invalidations are costly, and should be avoided as much as possible if speed performances are required. Gothic memory management, however, acts as expected (!). Processes can share any permanent segment in the Gothic system. The operating system keeps shared segments consistent. It is but up to the user to write programs that will use "cleverly" facilities the system is providing. Parallel clauses should be programmed in order to have a good distribution of calculus and data, and notice that these two must be quite related. Shared segments can be used for communication and synchronization. The "trick" is to used this facility only when really needed.

6.2 Final Remarks and Gothic Future

These first tests performed on the Gothic system show that Gothic consistency management protocol fits for parallel application executions provided these are showing a good locality. Of course, Gothic is not to be compared with any tightly coupled multiprocessor or any parallel computer. Gothic's purpose is rather to allow information sharing in a distributed system so that processes can communicate by sharing a permanent information. This is interesting for programming distributed applications and for synchronization between processes. Gothic seems to suit very well distributed data base applications as any distributed application with locality.

This first prototype of a Gothic system needs of course many improvements. The whole kernel code has to be checked for optimization and for getting better speed performances. The third step has to be achieved for the system to run on a LAN of MW. It is mainly inter-workstation communication that have to be installed. A priori, as Gothic is built on the "site" and not the workstation notion, it should be rather easily adapted.

This first version is a basis in which we get experiment and which validates our main ideas about SLS and consistency management. These benefits will be used for building Gothic next version. Other issues will now be considered. One among others is the object oriented issue. This concept is not fulfilled today in the Gothic kernel; capabilities, object-typing, garbage collection are under study. Another important problem is fault-tolerance. Gothic memory manager will be a reference for designing the memory manager of a new research project. This new project will consider fault-tolerance as a main issue and will integrate it in the system kernel itself. Beside this new project, fault-tolerance is now studied for the Gothic kernel too.

Acknowledgement

The author is grateful to Jean-Pierre Banâtre and Michel Banâtre for their help and advices in the design and implementation of Gothic Virtual Memory Management. She also would like to thank Bruno Rochat, Isabelle Puaut and Thierry Leconte who took essential parts in the achievement of this work.
This work has been made possible by Bull S.A. and Inria supports and grants.

References

[BANA 86] J.P.Banâtre, M.Banâtre, F.Ployette
The Concept of MultiFunction: a General Structuring Tool for Distributing Operating Systems. Proc. of the 6th International Conference on DCS, Cambridge, Mass., May 86.

[BANA 88a] J.P.Banâtre, M.Banâtre, G.Muller
Main Aspects of the Gothic Distributed System. in R.Speth (ed), Research into Networks and Distributed Applications EUTECO'88. Vienna, Austria, April 88. (North-Holland)

[BANA 88b] J.P.Banâtre, M.Banâtre, G.Muller
Ensuring Data Security with a Fast Stable Storage. Proc. of 4th International Conference on Data Engineering, Los Angeles, February 88.

[BANA 89] J.P.Banâtre, M.Banâtre, C.Morin
Implementing Atomic Rendez-vous within a Transactionnal Framework . Proc. of 8th Symposium on Reliable Distributed Systems, Seattle, October 89.

[BART 81] J.Bartlett
A NonStop Kernel. Proc. 8th on Operating System Principles, Pacific Grove, Ca, 81.

[BULL 85a] Doc. Bull
SPS7: Présentation générale. Janvier 85.

[BULL 85b] Doc. Bull
SPS7: Organes centraux. Modules de traitement 68010. Manuel d'utilisation. Janvier 85.

[BULL 86] Doc. Bull
SPS7: Organes centraux. Module de traitement 68020 mémoire virtuelle Manuel d'utilisation. Août 86.

[BULL 87] Doc. Bull
SPS7: Spart, système d'exploitation temps réel. Manuel de référence, Avril 87.

[CENS 78] L.M.Censier, P.Feautrier
A New Solution to Coherence in Multicache Systems. IEEE. Transactions on Computers, Vol.C-27, N0 12, December 78.

[DUBO 88] M.Dubois, C.Scheurich, F.A.Briggs
Synchronization, Coherence and Event Ordering in Multiprocessors. Computer. Surveys & Tutorial Series. IEEE. February 88.

[HOUD 81] M.E.Houdek, F.G.Soltis, R.L.Hoffman
IBM System/38 Support for Capability-Based Addressing. Proceedings of the 8th Annual Symposium on Computer Architecture. Minneapolis, Minnesota, ACM Sigarch, Vol.9, N0 3, May 81.

[LECE 88] P.LeCerten, P.Lecler, F.Ployette
Manuel d'utilisation du langage Polygoth. Rapport technique INRIA 88.

[LECL 89] P.Lecler
Une approche de la programmation des systèmes distribués fondée sur la fragmentation des données et des calculs, et sa mise en oeuvre dans le système Gothic. Thèse en Informatique, Université de Rennes 1, Septembre 89.

[LEVI 86] P.H.Levine
The Apollo Domain Distributed File System. NATO Advanced Study Institute. Izmir, Turkey, August 86.

[LI 86] K.Li
Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD Thesis. Yale University. September 86.

[MICH 89] B.Michel
Conception et Réalisation de la Mémoire Virtuelle de Gothic. Thèse en Informatique, Université de Rennes 1, Septembre 89.

[MOTO 86] Doc. Motorola
MC68851 Paged Memory Management Unit. User's Manual. 86.

[MULL 88] G.Muller
Conception et réalisation d'une machine multiprocesseur sûre de fonctionnement. Thèse en Informatique, Université de Rennes 1, Juin 88.

[RAMA 88] U.Ramachandran, H.Ahamad, A.Khalidi, M.Youssef
Unifying Synchronisation and Data Transfer in Maintaining Coherence of Distributed Shared Memory. Technical Report, Georgia Institute of Technology, June 88.

Liste des dernières publications internes parues à l'IRISA

PI 516 **COMMENT INTRODUIRE LA CONTIGUITE EN ANALYSE DES CORRESPONDANCES ? Application en segmentation d'image.**
Brigitte ESCOFIER, Habib BENALI, Kaddour BACHAR
Février 1990, 26 Pages.

PI 517 **MACHINE MODELING AND LOOP OPTIMIZATION FOR HORIZONTAL MICROCODED MACHINES**
François BODIN, François CHAROT
Février 1990, 24 Pages.

PI 518 **MULTISCALE SYSTEM THEORY**
Albert BENVENISTE, Ramine Nikoukhah, Alan S. Willsky.
Février 1990, 30 Pages.

PI 519 **PANDORE : A SYSTEM TO MANAGE DATA DISTRIBUTION**
Françoise ANDRE, Jean-Louis PAZAT, Henry THOMAS
Février 1990, 14 Pages.

PI 520 **SCHEDULING AFFINE PARAMETERIZED RECURRENCES BY MEANS OF VARIABLE DEPENDENT TIMING FUNCTIONS**
Christophe MAURAS, Patrice QUINTON
Sanjay RAJOPADHYE, Yannick SAOUTER
Février 1990, 14 Pages.

PI 521 **COMPUTABILITY OF RECURRENCE EQUATIONS**
Yannick SAOUTER, Patrice QUINTON
Février 1990, 28 Pages.

PI 522 **PROGRAMMING BY MULTISET TRANSFORMATION**
Jean-Pierre BANATRE, Daniel LE METAYER
Mars 1990, 26 Pages.

PI 523 **GOTHIC MEMORY MANAGEMENT : A MULTIPROCESSOR SHARED SINGLE LEVEL STORE**
Béatrice MICHEL
Mars 1990, 20 Pages.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique .

ISSN 0249-6399